UNIVERSITETET I OSLO
DET MATEMATISK-NATURVITENSKAPELIGE FAKULTET

Dagens tema

Programmering av x86

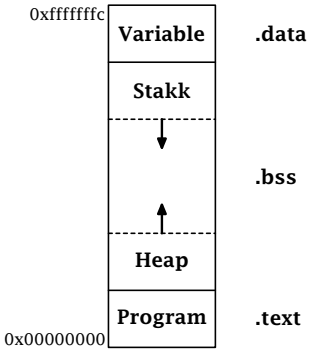
- Minnestrukturen
- Flytting av data
  - Endring av størrelse
- Aritmeriske operasjoner
  - Flagg
- Maskeoperasjoner
- Hopp
  - Tester
- Stakken
- Rutinekall
  - Kall og retur
  - Frie og opptatte registre
  - Dokumentasjon

Husk!
Alt er bare bit-mønstre!

INF2270

Minnestrukturen

Grovt sett ser minnet for hver process slik ut:



(**bbs** = «block started by symbol» fra IBM 704 ca 1950.)

INF2270

Flytting av data

(B&O'H-boken 3.4.2)
Instruksjonen mov kan flytte data til/fra

konstanter	$10
registre	%eax
navngitte variable	navn
lagerlokasjoner pekt på	0(%esp)

Men . . .

- Man kan ikke flytte *til* en konstant.
- Maksimalt én lagerlokasjon.

```
        .text
move:
        movl    $3,%eax
        movl    4(%esp),%eax
        movl    %eax,var
        ret

        .data
var:    .long 17   # En long med verdi 17
arr:    .fill  8   # 8 byte uten initialverdi
```

Variable
Man kan sette av plass til variable med spesifikasjonen .long eller .fill. De bør legges i .data.

INF2270

Byte, ord og langord

mov- finnes for -b («byte»), -w («word» = 2 byte) og -l («long» = 4 byte).

```
        .text
        movb    $0x12,%al
        movw    $0x1234,%ax
```

Kun de aktuelle delene av registrene endres.

Konvertering mellom størrelser
Fra større til mindre størrelser dropper man bare de bit-ene man ikke trenger.†

00000000	00000000	00000000	00000001

Fra mindre til større *unsigned* verdier er det bare å sette inn 0-er foran.

Fra mindre til større *signed* verdier finnes disse:

cbw	Utvider **%al** til **%ax**.
cwd	Utvider **%ax** til **%dx**:**%ax**.
cwde	Utvider **%ax** til **%eax**.
cdq	Utvider **%eax** til **%edx**:**%eax**.

† Hva om tallet er for stort? *Overflyt* vil vi ta for oss senere i kurset.

INF2270

Aritmetiske operasjoner

(B&O'H-boken 3.5.2 og 3.5.5)
Hittil kjenner vi

Addisjon:	addb	addw	addl
Økning:	incb	incw	incl
Subtraksjon:	subb	subw	subl
Senkning:	decb	decw	decl

I tillegg har vi

Negasjon:	negb	negw	negl
Multiplikasjon:	—	imulw	imull

Alle fungerer på registre og inntil én minnelokasjon.

INF2270

Multiplikasjon

I tillegg til den vanlige utgaven nevnt på forrige ark, finnes en versjon som jobber med faste registre:

mulb og imulb	**%al**	× *op* →	**%ax**
mulw og imulw	**%ax**	× *op* →	**%dx**:**%ax**
mull og imull	**%eax**	× *op* →	**%edx**:**%eax**

Fordelen med denne utgaven er at den finnes både for verdier *med* fortegn (imul- og versjonen på forrige ark) og *uten* fortegn (mul-).

Ulempen er at operand 2 kan være register eller minnelokasjon, men ikke konstant.

INF2270

Divisjon

Divisjon gir to svar (kvotient og rest). Den er også litt rar når det gjelder registerbruk:

divl og idivl	**%edx**:**%eax**	÷ *op* →	**%eax**	**%edx**
divw og idivw	**%dx**:**%ax**	÷ *op* →	**%ax**	**%dx**
divb og idivb	**%ax**	÷ *op* →	**%al**	**%ah**

Disse instruksjonen kan ikke dele på konstanter, kun på variable og registerverdier.

Eksempel

Denne funksjonen deler et tall med 10 og returnerer svaret og resten der de to adressene i parameter 2 og 3 angir.

```
.globl divl0
# C-signatur:   void divl0(int v, int *q, int *r).

divl0:
        movl    4(%esp),%eax    #       %eax = v.
        cdq                     # %edx:
        movl    $10,%ecx        # %ecx = 10.
        idivl   %ecx            # (%eax,%edx) = (%edx:%eax/10, %edx:%eax%10).

        movl    8(%esp),%ecx    # *q    = %eax.
        movl    %eax,(%ecx)     #       = %eax.
        movl    12(%esp),%ecx   # *r    = %edx.
        movl    %edx,(%ecx)     # Retur.
        ret
```

Testprogram

```
#include <stdio.h>

extern void div10 (int v, int *q, int *r);

int data[] = { 0, 19, 226, -17 };

int main (void)
{
  int data_len = sizeof(data)/sizeof(int), a1, a2, ix;

  for (ix = 0;  ix < data_len;  ++ix) {
    div10(data[ix], &a1, &a2);
    printf("%d/10 = %d, %d%%10 = %d\n",
           data[ix], a1, data[ix], a2);
  }
  return 0;
}
```

Kjøring

```
> gcc -m32 test-div10.c div10.s -o test-div10
> ./test-div10
0/10 = 0, 0%10 = 0
19/10 = 1, 19%10 = 9
226/10 = 22, 226%10 = 6
-17/10 = -1, -17%10 = -7
```

Advarsel!

Overflyt ved divisjon eller divisjon med 0 er ekstra farlig; hvis det skjer, får vi se følgende:

```
Floating point exception
```


Flagg (B&O'H-boken 3.6.1)

De fleste operasjonene har en bieffekt: visse egenskaper ved resultatet blir lagret i *flaggene*.

Z («Zero») settes til 1 når svaret er 0 (og til 0 ellers).

S («Sign») settes lik øverste bit i svaret. (Om vi regner med *signed* tall, er dette et tegn på at tallet er negativt.)

C («Carry» = mente) settes lik den menteoverføringen som skjedde øverst i resultatet.

O («Overflow») settes om svaret var for stort.

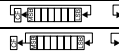
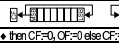
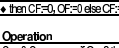
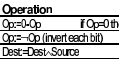
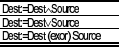
P («Parity») settes om *laveste byte* har et partall antall 1-bit.

Inneholder flaggene nyttig informasjon?
Av og til, men ikke alltid.

intel Assembler 80186 and higher **CodeTable 1/2**

TRANSFER Name	Comment	Code	Operation	O	D	I	T	S	Z	A	P	C
MOV	Move (copy)	MOV Dest,Source	Dest:=Source									
XCHG	Exchange	XCHG Op1,Op2	Op1:=Op2, Op2:=Op1									
STC	Set Carry	STC	CF:=1									1
CLC	Clear Carry	CLC	CF:=0									0
CMC	Complement Carry	CMC	CF:=~CF									±
STD	Set Direction	STD	DF:=1 (string op's downwards)		1							
CLD	Clear Direction	CLD	DF:=0 (string op's upwards)		0							
STI	Set Interrupt	STI	IF:=1			1						
CLI	Clear Interrupt	CLI	IF:=0			0						
PUSH	Push onto stack	PUSH Source	DEC SP, [SP]:=Source									
PUSHF	Push flags	PUSHF	O, D, I, T, S, Z, A, P, C 286+: also NT, IOPL									
PUSHA	Push all general registers	PUSHA	AX, CX, DX, BX, SP, BP, SI, DI									
POP	Pop from stack	POP Dest	Dest:=[SP], INC SP									
POPF	Pop flags	POPF	O, D, I, T, S, Z, A, P, C 286+: also NT, IOPL	±	±	±	±	±	±	±	±	±
POPA	Pop all general registers	POPA	DI, SI, BP, SP, BX, DX, CX, AX									
CBW	Convert byte to word	CBW	AX:=AL (signed)									
CWD	Convert word to double	CWD	DX:AX:=AX (signed)	±				±	±	±	±	±
CWDE	Conv word extended double	CWDE 386	EAX:=AX (signed)									
IN *i*	Input	IN Dest, Port	AL/AX/EAX = byte/word/double of specified port									
OUT *i*	Output	OUT Port, Source	Byte/word/double of specified port := AL/AX/EAX									

i for more information see instruction specifications — Flags: ±=affected by this instruction ?=undefined after this instruction

ARITHMETIC Name	Comment	Code	Operation	O	D	I	T	S	Z	A	P	C
ADD	Add	ADD Dest,Source	Dest:=Dest+Source	±				±	±	±	±	±
ADC	Add with Carry	ADC Dest,Source	Dest:=Dest+Source+CF	±				±	±	±	±	±
SUB	Subtract	SUB Dest,Source	Dest:=Dest-Source	±				±	±	±	±	±
SBB	Subtract with borrow	SBB Dest,Source	Dest:=Dest-(Source+CF)	±				±	±	±	±	±
DIV	Divide (unsigned)	DIV Op	Op=byte: AL:=AX / Op AH:=Rest	?				?	?	?	?	?
DIV	Divide (unsigned)	DIV Op	Op=word: AX:=DX:AX / Op DX:=Rest	?				?	?	?	?	?
DIV 386	Divide (unsigned)	DIV Op	Op=doublew.: EAX:=EDX:EAX / Op EDX:=Rest	?				?	?	?	?	?
IDIV	Signed Integer Divide	IDIV Op	Op=byte: AL:=AX / Op AH:=Rest	?				?	?	?	?	?
IDIV	Signed Integer Divide	IDIV Op	Op=word: AX:=DX:AX / Op DX:=Rest	?				?	?	?	?	?
IDIV 386	Signed Integer Divide	IDIV Op	Op=doublew.: EAX:=EDX:EAX / Op EDX:=Rest	?				?	?	?	?	?
MUL	Multiply (unsigned)	MUL Op	Op=byte: AX:=AL*Op if AH=0 ♦	±				?	?	?	?	±
MUL	Multiply (unsigned)	MUL Op	Op=word: DX:AX:=AX*Op if DX=0 ♦	±				?	?	?	?	±
MUL 386	Multiply (unsigned)	MUL Op	Op=double: EDX:EAX:=EAX*Op if EDX=0 ♦	±				?	?	?	?	±
IMUL *i*	Signed Integer Multiply	IMUL Op	Op=byte: AX:=AL*Op if AL sufficient ♦	±				?	?	?	?	±
IMUL	Signed Integer Multiply	IMUL Op	Op=word: DX:AX:=AX*Op if AX sufficient ♦	±				?	?	?	?	±
IMUL 386	Signed Integer Multiply	IMUL Op	Op=double: EDX:EAX:=EAX*Op if EAX sufficient ♦	±				?	?	?	?	±
INC	Increment	INC Op	Op:=Op+1 (Carry not affected !)	±				±	±	±	±	
DEC	Decrement	DEC Op	Op:=Op-1 (Carry not affected !)	±				±	±	±	±	
CMP	Compare	CMP Op1,Op2	Op1-Op2	±				±	±	±	±	±
SAL	Shift arithmetic left (=SHL)	SAL Op,Quantity		*i*				±	±	?	±	±
SAR	Shift arithmetic right	SAR Op,Quantity		*i*				±	±	?	±	±
RCL	Rotate left through Carry	RCL Op,Quantity		*i*								±
RCR	Rotate right through Carry	RCR Op,Quantity		*i*								±
ROL	Rotate left	ROL Op,Quantity		*i*								±
ROR	Rotate right	ROR Op,Quantity		*i*								±

i for more information see instruction specifications ♦ then CF:=0, OF:=0 else CF:=1, OF:=1

LOGIC Name	Comment	Code	Operation	O	D	I	T	S	Z	A	P	C
NEG	Negate (two-complement)	NEG Op	Op:=0-Op if Op=0 then CF:=0 else CF:=1	±				±	±	±	±	±
NOT	Invert each bit	NOT Op	Op:=~Op (invert each bit)									
AND	Logical and	AND Dest,Source	Dest:=Dest\Source	0				±	±	?	±	0
OR	Logical or	OR Dest,Source	Dest:=Dest\/Source	0				±	±	?	±	0
XOR	Logical exclusive or	XOR Dest,Source	Dest:=Dest (exor) Source	0				±	±	?	±	0
SHL	Shift logical left (=SAL)	SHL Op,Quantity		*i*				±	±	?	±	±
SHR	Shift logical right	SHR Op,Quantity		*i*				±	±	?	±	±

Maskeoperasjoner

(B&O'H-boken 3.5.2)

Maskeoperasjonene brukes til å sette eller nulle ut bit i henhold til et gitt mønster (en såkalt *maske*).

Maske-AND

Denne operasjonen *nuller ut* de bit som ikke er markert i masken.†

	0	1	0	1	0	1	0	1
andb	0	0	0	0	1	1	1	1
=	0	0	0	0	0	1	0	1

Denne operasjonen er tilgjengelig i C og heter der &.

NB! Det er stor forskjell på & (maske-AND eller bit-AND) og && (logisk AND) i C:

```
1 & 4 == 0
1 && 4 == 1
```

† Siden operasjonen er symmetrisk, er det vilkårlig hvilken operand som betraktes som maske og hvilken som er data.

Maske-OR

Denne operasjonen *setter* de bit som er markert i masken.

	0	1	0	1	0	1	0	1
orb	0	0	0	0	1	1	1	1
=	0	1	0	1	1	1	1	1

Denne operasjonen er tilgjengelig i C og heter der |.

INF2270

Maske-NOT

Denne operasjonen snur *alle* bit-ene.

	0	1	0	1	1	1	1	1
notb								
=	1	0	1	0	0	0	0	0

Den finnes også i C og heter der ~.

INF2270

Maske-XOR

Denne operasjonen *snur* bare de bit som er markert i masken.

	0	1	0	1	0	1	0	1
xorb	0	0	0	0	1	1	1	1
=	0	1	0	1	1	0	1	0

Denne operasjonen kalles også ofte «logisk addisjon».

Den er tilgjengelig i C og heter der ^.

INF2270

Hopp (B&O'H-boken 3.6.3–5)

Instruksjonen for å hoppe heter jmp.

```
        jmp dit

dit:
```

Betinget hopp

Man kan angi at flaggene skal avgjøre om man skal hoppe.

```
        jz      dit     # Hopp om Z(ero)
        jnz     dit     # Hopp om ikke Z
        jc      dit     # Hopp om C(arry)
        jnc     dit     # Hopp om ikke C
        js      dit     # Hopp om S(ign)
        jns     dit     # Hopp om ikke S
        jo      dit     # Hopp om O(verflow)
        jno     dit     # Hopp om ikke O
        jp      dit     # Hopp om P(arity)
        jnp     dit     # Hopp om ikke P
```

INF2270

Testing

Flaggene kan settes som følge av vanlige instruksjoner:

```
.globl abs2
abs2:   movl    4(%esp),%eax
        addl    8(%esp),%eax
        jns     ret2
        negl    %eax
ret2:   ret
```

Alternativt kan vi eksplisitt sjekke to verdier mot hverandre med instruksjonen cmp-:

```
.globl abs1
abs1:   movl    4(%esp),%eax
        cmpl    $0,%eax
        jns     ret1
        negl    %eax
ret1:   ret
```

INF2270

Hva er riktige flagg å sjekke på ved for eksempel **%eax**$\leqslant -17$? Heldigvis finnes spesielle varianter som er enklere å bruke:

Verdier *med* fortegn

```
        je      dit     # Hopp ved =  (= Z)
        jne     dit     # Hopp ved != (= ~Z)
        jl      dit     # Hopp ved <  (= S)
        jle     dit     # Hopp ved <= (= Z || S)
        jg      dit     # Hopp ved >  (= ~Z && ~S)
        jge     dit     # Hopp ved >= (= ~S)
```

Verdier *uten* fortegn

```
        je      dit     # Hopp ved =  (= Z)
        jne     dit     # Hopp ved != (= ~Z)
        jb      dit     # Hopp ved <  (= C)
        jbe     dit     # Hopp ved <= (= Z || C)
        ja      dit     # Hopp ved >  (= ~C && ~Z)
        jae     dit     # Hopp ved >= (= ~C)
```

INF2270

Eksempel

Denne funksjonen finner det minste av to tall:

```
min2:   movl    4(%esp),%eax
        cmpl    8(%esp),%eax
        jle     ret
        movl    8(%esp),%eax
ret:    ret
```

NB!

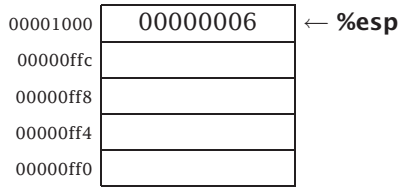
Testen blir *omvendt* i Linux siden operandene kommer i en annen rekkefølge!

INF2270

Stakken

Stakken er veldig sentral i x86-arkitekturen. Den benyttes til

- rutinekall
- parameteroverføring
- lagring av mellomresultater
- plass til lokale variable

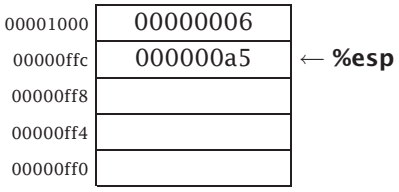


Av historiske grunner vokser stakken mot *lavere* adresser.

INF2270

Å legge elementer på stakken
Instruksjonene pushw og pushl legger verdier på stakken:

```
    pushl   $0x000000a5
```



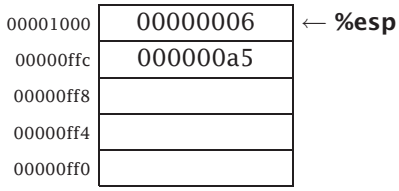
Legg merke til at vi kan få tak i alle elementene på stakken:

```
    movl    0(%esp),%eax    # Toppen
    movl    4(%esp),%eax    # Nest øverst
```

INF2270

Å fjerne elementer fra stakken
Til dette brukes popw og popl:

```
    popl    %eax
```



Verdiene blir ikke fysisk fjernet.

INF2270

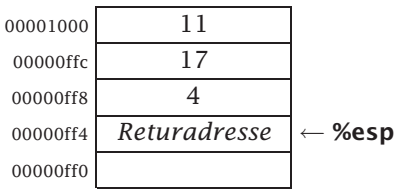
Rutiner (B&O'H-boken 3.7.2-4)
Ved et rutinekall skjer følgende:

1. Parametrene beregnes og legges på stakken *bakfra!*
2. Instruksjonen call fungerer som en jmp men legger adressen til neste instruksjon på stakken.

Kallet

```
    f(4, 17, 11);
```

vil gi denne stakken:



Ved retur vil ret fjerne returadressen fra stakken og hoppe dit.

(Det er opp til kalleren å fjerne parametrene fra stakken.)

INF2270

Registerbruk
Hvilke registre kan vi endre i en funksjon uten å ødelegge for kalleren?

Frie registre
Konvensjonen er at

%eax, **%ecx** og **%edx**

er *frie registre* («caller save»).

Bundne registre
De andre registrene er *bundne registre* («callee save»). Om de endres, må man ta vare på den opprinnelige verdien og sette denne tilbake før retur.

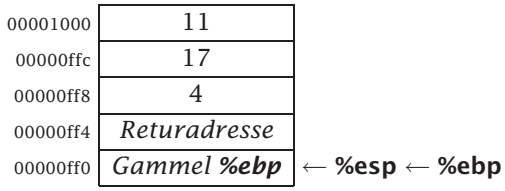
INF2270

En forbedring
Hittil har vi hentet parametrene som 4(%esp), 8(%esp), . . .

Men hva om vi ønsker å lagre mellomresultater på stakken? Da må adresseringen endres!

Løsningen er å bruke et eget register **%ebp** til å peke på parametrene:

```
    pushl   %ebp
    movl    %esp,%ebp
```



Nå er paramtrene tilgjengelige som 8(%ebp), 12(%ebp), . . .

Retur må nå gjøres slik:

```
    popl    %ebp
    ret
```

INF2270

Dokumentasjon

Målet med dokumentasjon er man skal kunne få vite alt man trenger for å bruke en funksjon ved å lese dokumentasjonen. Dette inkluderer:

❶ funksjonens navn

❷ hva den gjør (kort fortalt)

❸ parametrene

I tillegg kan det være nyttig å vite hva de ulike registrene brukes til når man skal lese koden.

```
                .globl mystrlen

# Name:  mystrlen.
# Synopsis:     Beregner antall tegn i en tekst.
# C-signatur:   int mystrlen (char *s)
# Registre:     EAX:    len
#               ECX:    s

mystrlen:
        pushl   %ebp            # Standard
        movl    %esp,%ebp       # funksjonsstart.

        movl    8(%esp),%ecx    # %ecx = s.
        movl    $0,%eax         # %eax = 0.
loop:   cmpb    $0, (%ecx)      # while (%ecx
        je      exit            #             !=0) {
        incl    %eax            #   ++len.
        incl    %ecx            #   ++s.
        jmp     loop            # }

exit:   popl    %ebp            # Standard retur.
        ret                     # return len.
```

INF2270